Graph Mining - CSF426

Lab Session - 15

Date - 21/10/2023

Marks - 10

Instructor IC - Vinti Agarwal

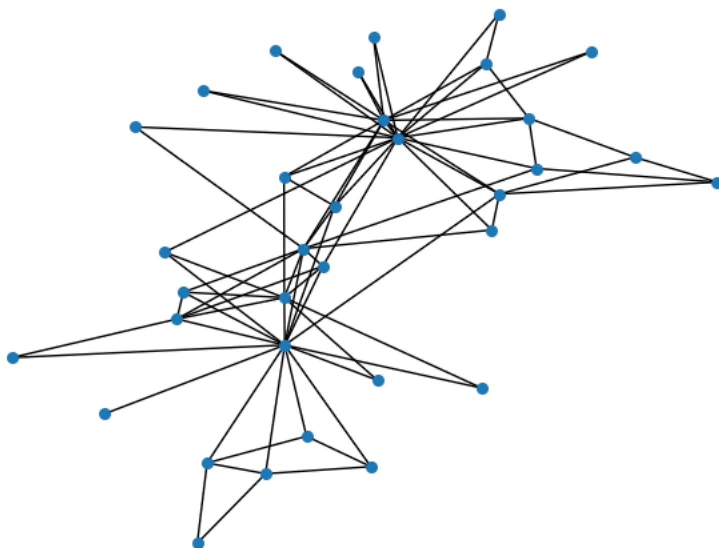
Objective:

This programming exercise is designed to compare the approach of Spectral Clustering and Girvan-Newman Clustering. The code for spectral clustering is provided below. You are required to implement code for Girvan-Newman clustering algorithm. And then comprare the clusters by plotting them.

The cells in which students are supposed to write thier code are marked as TO-DO

```
import networkx as nx
from networkx.algorithms import community
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh
from collections import deque
```

```
# Karate club graph with nodes = 34 and edges = 78
G = nx.karate_club_graph()
nx.draw(G, node_size = 30)
```



```
labels = nx.get_node_attributes(G, "club")
y = []
for key in labels.keys() :
    y.append(labels[key])
```

Spectral Clustering: This section implements spectral clustering. As a result, four different clusters are formed.

```
# Laplacian and eigenvalues for whole graph
L = nx.laplacian_matrix(G).toarray()
A = nx.adjacency_matrix(G).toarray()
# D = L + A
```

```
val, vec = eigh(L)
fiedler = vec[:,1]
val
```

```
#partition graph G based on fiedler vector
P_spec1 = np.where( fiedler < 0)
P_spec2 = np.where( fiedler > 0)
P_spec1[0], P_spec2[0]
```

```
# getting the labels for nodes in each cluster
y_P1 = np.array(y)[list(map(list, P_spec1))[0]]
y_P2 = np.array(y)[list(map(list, P_spec2))[0]]
```

Purity function()

```
#  Calculate the purity, a measurement of quality for the clustering results.
# Each cluster is assigned to the class which is most frequent in the
# cluster.  Using these classes, the percent accuracy is then calculated.

# Returns:
#   A number between 0 and 1.  Poor clusterings have a purity close to 0
#   while a perfect clustering has a purity of 1.
def purity(labels):
  count1, count2 = 0,0
  for label in labels:
    if label == 'Mr. Hi':
      count1 = count1+1
    else:
      count2 = count2+1
  # print(count1, count2)
  purity = max(count1, count2)/len(labels)
  return purity
```

```
print("Purity of cluster1 ", purity(y_P1))
print("Purity of cluster1 ", purity(y_P2))
```

```
#subgraph of Graph G based on values less than 0 in fiedler vector
sub_1 = nx.subgraph(G, P_spec1[0])
W_1 = nx.adjacency_matrix(sub_1)
L_sub_1  = nx.laplacian_matrix(sub_1).toarray()
D_sub_1 = L_sub_1 + W_1
val_1, vec_1 = eigh(L_sub_1)
fiedler_1 = vec_1[:,1]
fiedler_1
```

```
#partitioning the subgraph_1 based on fiedler_1
Ind1_1 = np.where( fiedler_1 < 0)
Ind1_2 = np.where( fiedler_1 > 0)

P_spec11 = []
for i in Ind1_1[0]:
  P_spec11.append(P_spec1[0][i])  #append common nodes of P1
P_spec12 = []
for i in Ind1_2[0]:
  P_spec12.append(P_spec1[0][i])  #ppend common nodes of P1
Ind1_2
```

```
#subgraph of Graph G based on values greater than 0 in fiedler vector
sub_2 = nx.subgraph(G, P_spec2[0])
W_2 = nx.adjacency_matrix(sub_2)
L_sub_2  = nx.laplacian_matrix(sub_2).toarray()
D_sub_2 = L_sub_2 + W_2
val_2, vec_2 = eigh(L_sub_2)
fiedler_2 = vec_2[:,1]
```

```
Ind2_1 = np.where( fiedler_2 < 0)
Ind2_2 = np.where( fiedler_2 > 0)
P_spec21 = []
for i in Ind2_1[0]:
  P_spec21.append(P_spec2[0][i])
P_spec22 = []
for i in Ind2_2[0]:
  P_spec22.append(P_spec2[0][i])
```

```
# Clusters generated by Spectral clustering approach
P_spec11,P_spec12,P_spec21,P_spec22


# getting the labels for nodes in each cluster
y_P11 = np.array(y)[P_spec11]
y_P12 = np.array(y)[P_spec12]
y_P21 = np.array(y)[P_spec21]
y_P22 = np.array(y)[P_spec22]


print("Purity of cluster1 ", purity(y_P11))
print("Purity of cluster1 ", purity(y_P12))
print("Purity of cluster1 ", purity(y_P21))
print("Purity of cluster1 ", purity(y_P22))
```

GN clustering

Implement Girvan-Newman algorithm using following steps:

1. Start BFS at each node.
2. Compute edge weights and node weights.
3. Calculate the edge betweenness for each edge in graph.
4. Remove the edge with highest edge betweenness.
5. Repeat the above steps 1-4 for each subgraph now to get final 4 clusters.

```
def bfs(graph, start):
    """
    Perform Breadth-First Search (BFS) from a given start node.

    Args:
    graph: Dictionary, where keys are nodes and values are lists of neighbors.
    start: Starting node for BFS.

    Returns:
    distances: Dictionary of shortest distances from start node to every other node.
    parents: Dictionary of parent nodes in the shortest path tree rooted at start node.
    """
    visited = set()
    queue = deque([start])
    distances = {start: 0}
    parents = {node: [] for node in graph}  # Initialize parents dictionary with empty lists for each node

    while queue:
        node = queue.popleft()
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited and neighbor not in queue:
                queue.append(neighbor)
                distances[neighbor] = distances[node] + 1
                parents[neighbor].append(node)
            elif neighbor in queue and distances[neighbor] == distances[node] + 1:
                parents[neighbor].append(node)
    # print(distances)
    # print(parents)
    return distances, parents
```

TO-DO

```
def compute_weights(______ , ______ , ______,dict_edge_weights):
    """
    Compute node weights and edge weights.

    Args:
    1. Dictionary, where keys are nodes and values are lists of neighbors.
    2. Dictionary of shortest distances from start node to every other node.
    3. Dictionary of parent nodes in the shortest path tree rooted at start node.
    4. Dictionary to store the sum up of edge weights corresponding to each edge in constructed BFS tree. For each edge, initialized wi

    Returns:
    edge_weights: Dictionary of edge weights.
```

```
    """
    node_weights = {node: 1 for node in graph}  # Initialize each node weight to 1
    edge_weights = {}                            # intitialization

    ## WRITE CODE HERE
```

TO-DO

```
def get_max_edge_betweenness(______):
    # INPUT: Dictionary that containes the sum of all edge weights corresponding to each edge obtained after performing BFS on all nodes
    # OUTPUT: Return all the node pairs having maximum edge betweenness


    return edges_max_betweenness
```

TO-DO

```
def girvan_newman(G,num_clusters):
  edges = G.edges()
  dict_edge_weights = {edge: 0 for edge in edges}
  clusters = []
  while len(clusters) < num_clusters:
    for node in G.nodes():
      distances, parents = bfs(G, node)
      compute_weights(_____, ______, _____, dict_edge_weights)   # pass as argument the graph and other supporting dictionaries.

    max_betwn_edges = get_max_edge_betweenness(_________)   # pass as argument the dictionary that contains the sum of all the edge weig
    G.remove_edges_from(max_betwn_edges)

    clusters = list(nx.connected_components(G))
  return clusters


# main part
num_clusters = 2
clutsers1 = girvan_newman(G.copy(), num_clusters)   # Implements GN algorithm on whole graph, output will be a list containing two clust
```

TO-DO

compute the purity of clusters by using purity() function defined in initial cells above and corresponding labels of nodes in each cluster

```
# write the code to compute the purity of each generated cluster.


# Again run GN algorithm on each cluster to further get more clusters
G2 = nx.subgraph(G,clutsers1[0])                    # create a new subgraph from the nodes in first cluster in clusters1
clutsers2 = girvan_newman(G2.copy(), num_clusters)  # Implement GN algorithm on this new subgraph to get further two more clusters

G3 = nx.subgraph(G,clutsers1[1])                    # create a new subgraph from the nodes in second cluster in clusters1
clutsers3 = girvan_newman(G3.copy(), num_clusters)  # Implement GN algorithm on this new subgraph to get further two more clusters

final_clusters = clutsers2 + clutsers3              #concatenate the clusters obtained from implmenting clustering on two subgraphs to g
final_clusters
```

Expected output:

[{0, 1, 3, 7, 11, 12, 13, 17, 19, 21},

{4, 5, 6, 10, 16},

{2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33},

{26}]

```
# Nenaming the clusters found
PG11 = final_clusters[0]
PG12 = final_clusters[1]
PG21 = final_clusters[2]
PG22 = final_clusters[3]
```

TO-DO

Again compute the purity of clusters by using purity() function and corresponding labels of nodes in each cluster

```
# write the code to compute purity
```

Plotting the graph partitions

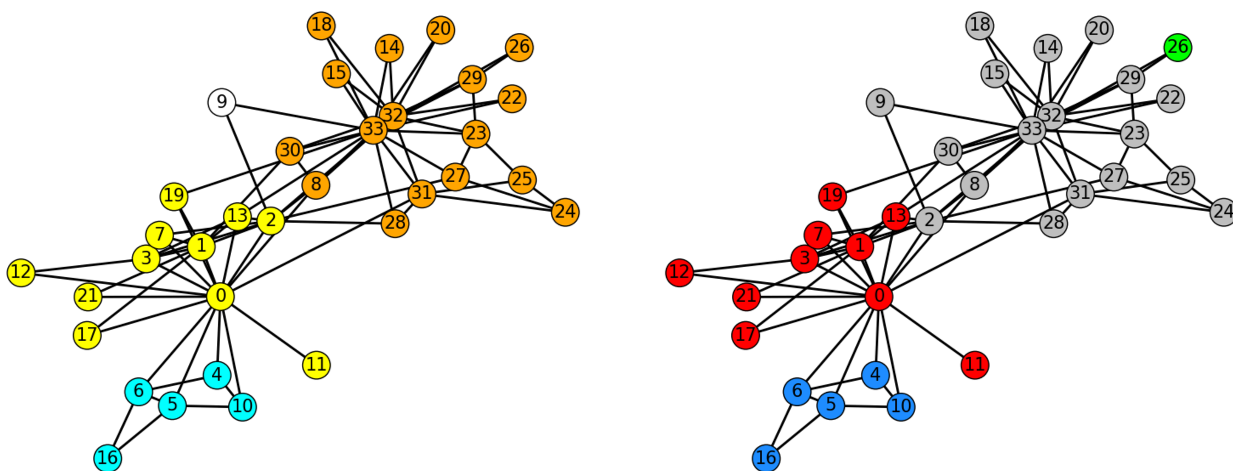
```
color_map_sp = np.empty(nx.number_of_nodes(G), dtype=object)
for node in P_spec11:
  color_map_sp[node] = "white"
for node in P_spec12:
  color_map_sp[node] = "orange"
for node in P_spec21:
  color_map_sp[node] = "cyan"
for node in P_spec22:
  color_map_sp[node] = "yellow"

options = {
    "font_size": 15,
    "node_size": 550, "node_color": color_map_sp, "linewidths": 1,
    "edgecolors": "black",
    "width": 2,
    "with_labels": True
}

color_map_gn = np.empty(nx.number_of_nodes(G), dtype=object)
for node in PG11:
    color_map_gn[node] = 'red'
for node in PG12:
    color_map_gn[node] = 'dodgerblue'
for node in PG21:
    color_map_gn[node] = 'silver'
for node in PG22:
    color_map_gn[node] = 'lime'

options_gn = {
    "font_size": 15,
    "node_size": 550, "node_color": color_map_gn, "linewidths": 1,
    "edgecolors": "black",
    "width": 2,
    "with_labels": True
}
```

Expected output:



```
position = nx.spring_layout(G)
plt.figure(figsize=(20,8))
plt.subplots_adjust( wspace=0.0, hspace=0.0)
plt.subplot(1,2,1)
nx.draw(G, position, **options)
```

```
plt.subplot(1,2,2)
nx.draw(G, position, **options_gn)
```

Comparing performance of both clustering approaches

```
# partition quality gives coverage and performance of a partition in graph. Coverage of cluster is ratio of intra-cluster edges to all e
# edges plus inter cluster non-edges divided by total number of edges. high coverage better partition.
print(community.partition_quality(G, [P_spec11,P_spec12,P_spec21,P_spec22]))
print(community.partition_quality(G, [PG11,PG12,PG21,PG22]))
```

```
# modularity is the measure of compring the inter and intra cluster connections. (high modularity means more dense intra cluster edges a
print(community.modularity(G, [P_spec11,P_spec12,P_spec21,P_spec22]))
print(community.modularity(G, [PG11,PG12,PG21,PG22]))
```